System design: Uber

Suresh Podeti
7 min read · Oct 13, 2023

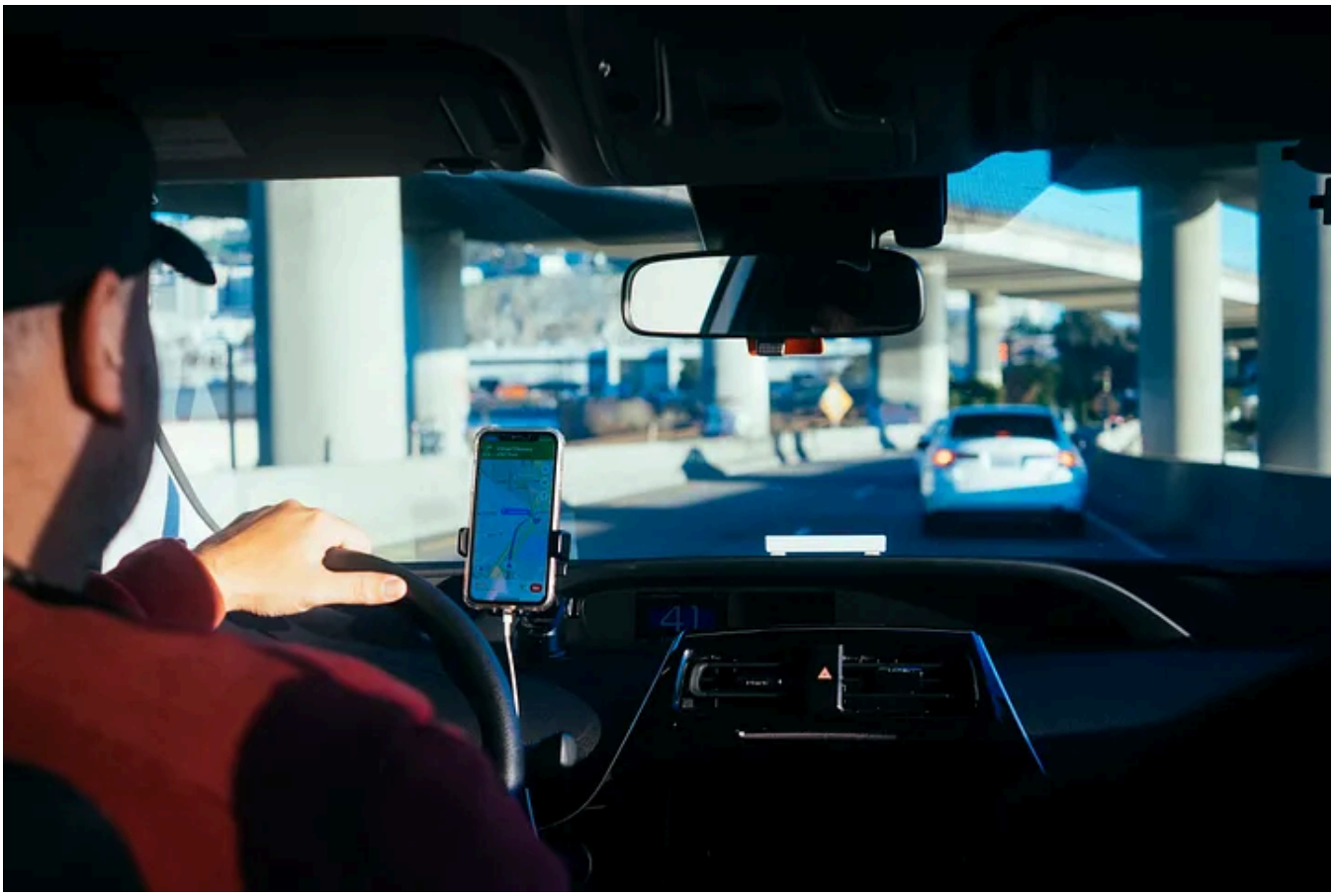

Photo by Paul Hanaoka on Unsplash

Introduction

Uber is an application that provides **ride-hailing services** to its users. Anyone who needs a ride can register and book a vehicle to travel from source to destination. Anyone who has a vehicle can register as a driver and take riders to their destination. Drivers and riders can communicate through the Uber app on their smartphones.

Requirements

Functional

- **Update driver location**: The driver is a moving entity, so the driver's location should be automatically updated at regular intervals.
- **Find nearby drivers**: The system should find and show the nearby available drivers to the rider.
- **Request a ride**: A rider should be able to request a ride, after which the nearest driver should be notified about the rider's requests.
- **Manage payments:** At the start of the trip, the system must initiate the payment process and manage the payments.
- **Show driver estimated time of arrival (ETA):** The rider should be able to see the estimated time of arrival of the driver.
- **Confirm pickup:** Drivers should be able to confirm that they have picked up the rider.
- **Show trip updates**: Once a driver and a rider accept a ride, they should be able to constantly see trip updates like ETA and current location until the trip finishes.
- **End the trip:** The driver marks the journey complete upon reaching the destination, and they then become available for the next ride.

Non-functional

- **Availability:** The system should be highly available. The downtime of even a fraction of a second can result in a trip failure, in the driver being unable to locate the rider, or in the rider being unable to contact the driver.
- **Scalability:** The system should be scalable to handle an ever-increasing number of drivers and riders with time.
- **Reliability:** The system should provide fast and error-free services. Ride requests and location updates should happen smoothly.
- **Consistency:** The system must be **strongly consistent**. The drivers and riders in an area should have a consistent view of the system.

- **Fraud detection:** The system should have the ability to detect any fraudulent activity related to payment.

Design

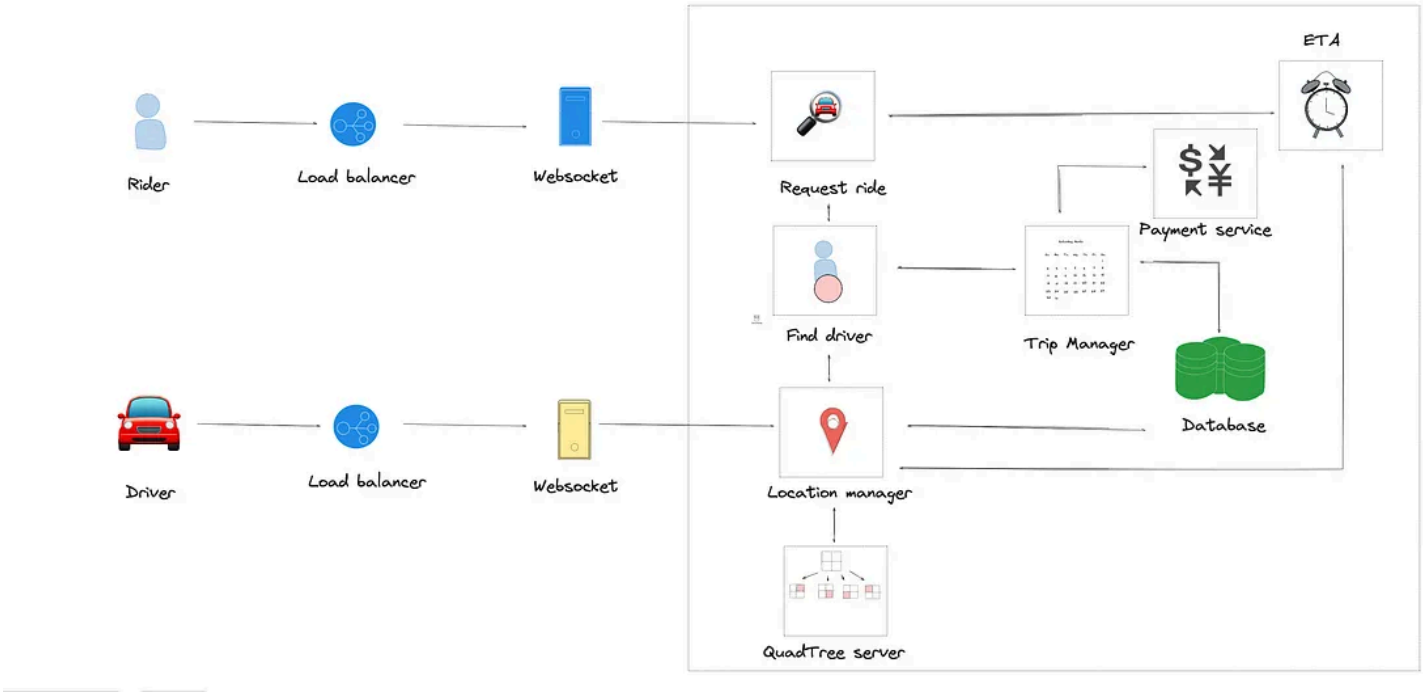

Fig 1.0: Uber high level design

Components

Location manager

This **service shows the nearby drivers** to the riders when they open the application. This service also **receives location updates from the drivers every four seconds.** The location of drivers is then communicated to the quadtree service to determine which segment the driver belongs to on a map. The **location manager saves the last location** of all drivers in a database and saves the route followed by the drivers on a trip.

Quadtree service

The **quadtree map service** updates the location of the drivers. The main problem is how we deal with finding nearby drivers efficiently.

Quadtrees help to divide the map into segments. If the number of drivers exceeds a certain limit, for example, 500, then we split that segment into four more child nodes and divide the drivers into them.

Each leaf node in quadtrees contains segments that can't be divided further. We can use the same quadtrees for finding the drivers. The most significant difference we have now is that our quadtree wasn't designed with regular upgrades in consideration. So, we have the following issues with our dynamic segment solution.

We must update our data structures to point out that all active drivers update their location every four seconds. **It takes a longer amount of time to modify the Quadtree whenever a driver's position changes.** To identify the driver's new location, we must first find a proper grid depending on the driver's previous position. If the new location doesn't match the current grid, we should remove the driver from the current grid and shift it to the correct grid.

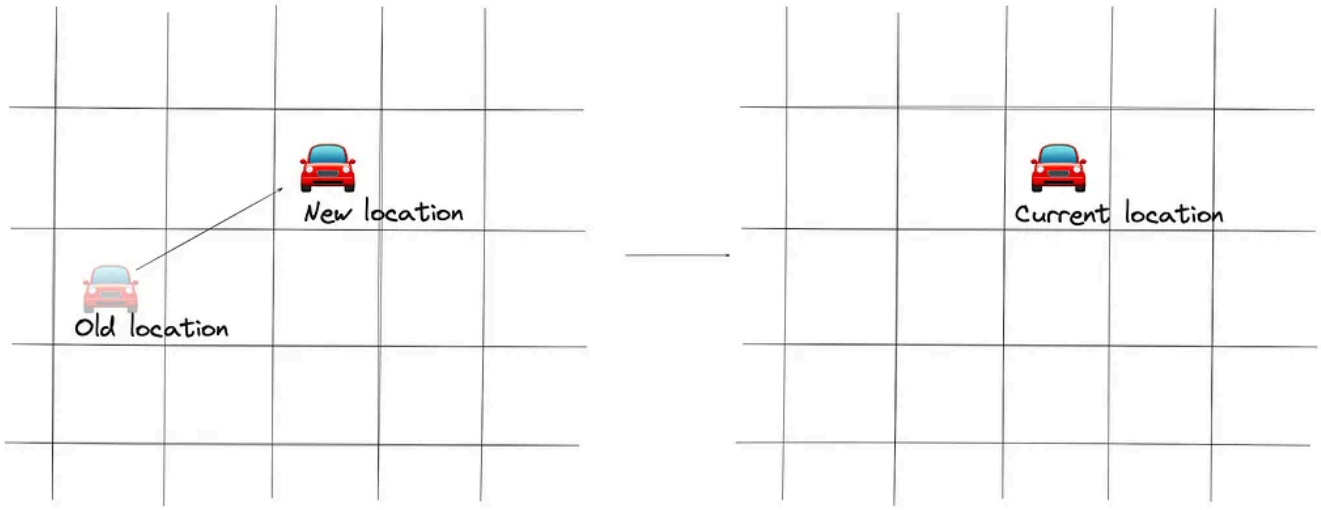

Fig 2.0: Updating the new current location

To overcome the above problem, we can use a hash table to store the latest position of the drivers and update our quadtree occasionally, say after 10–15 seconds. We can update the driver's location in the quadtree around every 15 seconds instead of four seconds, and we use a hash table that updates every four seconds and reflects the drivers' latest location. By doing this, we use fewer resources and time.

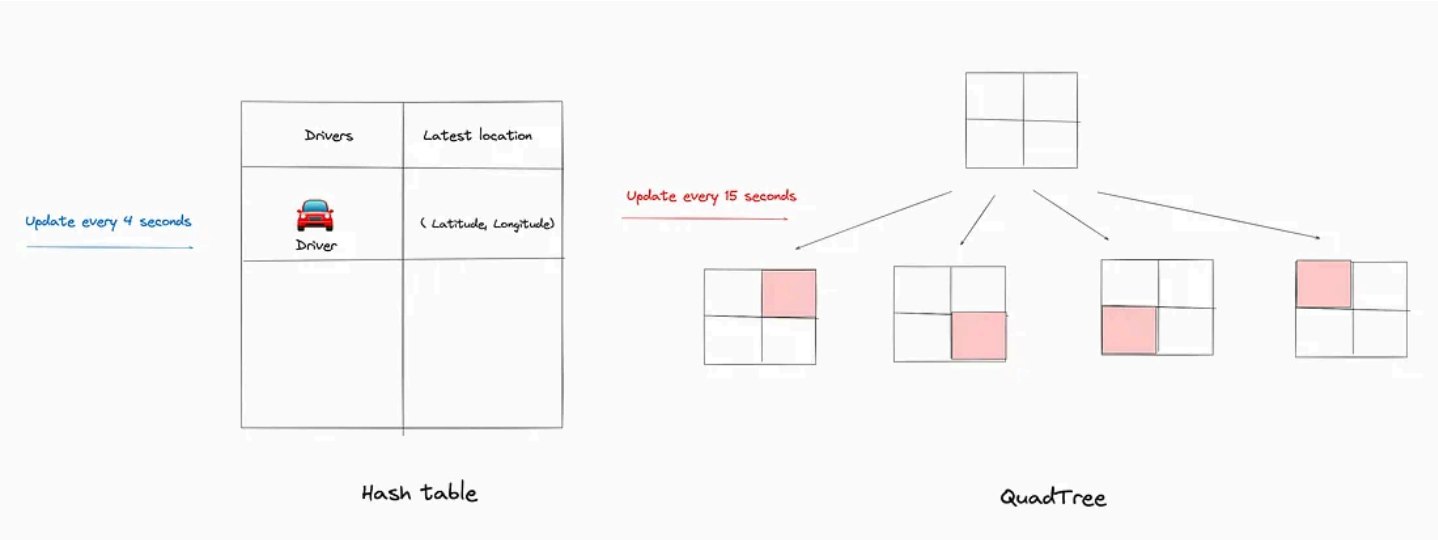

Fig 3.0: Hash table is updated every 4 secs, and Quadtree updated for every 15 secs

A million drivers need to send the updated location every four seconds. A million requests to the quadtree service affects how well it works. For this, we first store the updated location in the hash table stored in Redis. Eventually, these values are copied into the persistent storage every 10–15 seconds.

Request ride

The rider contacts the **request ride** service to request a ride. The rider adds the drop-off location here. The request vehicle service then communicates with the find driver service to book a vehicle and get the details of the vehicle using the location manager service.

Find driver

The **find driver** service finds the driver who can complete the trip. It sends the information of the selected driver and the trip information back to the request vehicle service to communicate the details to the rider. The find driver service also contacts the trip manager to manage the trip information.

Trip manager

The **trip manager** service manages all the trip-related tasks. It creates a trip in the database and stores all the information of the trip in the database.

ETA service

The **ETA service** deals with the estimated time of arrival. It shows riders the pickup ETA when their trip is scheduled. This service considers factors such as route and traffic. The two basic components of predicting an ETA given an origin and destination on a road network are the following:

- Calculate the shortest route from origin to destination.
- Compute the time required to travel the route.

To identify the shortest path between source and destination, we can utilize routing algorithms such as Dijkstra's algorithm. However, Dijkstra, or any other algorithm that operates on top of an unprocessed graph, is quite slow for such a system.

Therefore, this method is impractical at the scale at which these ride-hailing platforms operate.

To resolve these issues, we can split the whole graph into partitions. We **preprocess the optimum path inside partitions** using contraction hierarchies and deal with just the partition boundaries. This strategy can considerably reduce the time complexity since it partitions the graph into layers of tiny cells that are largely independent of one another. The preprocessing stage is executed in parallel in the partitions when necessary to increase speed.

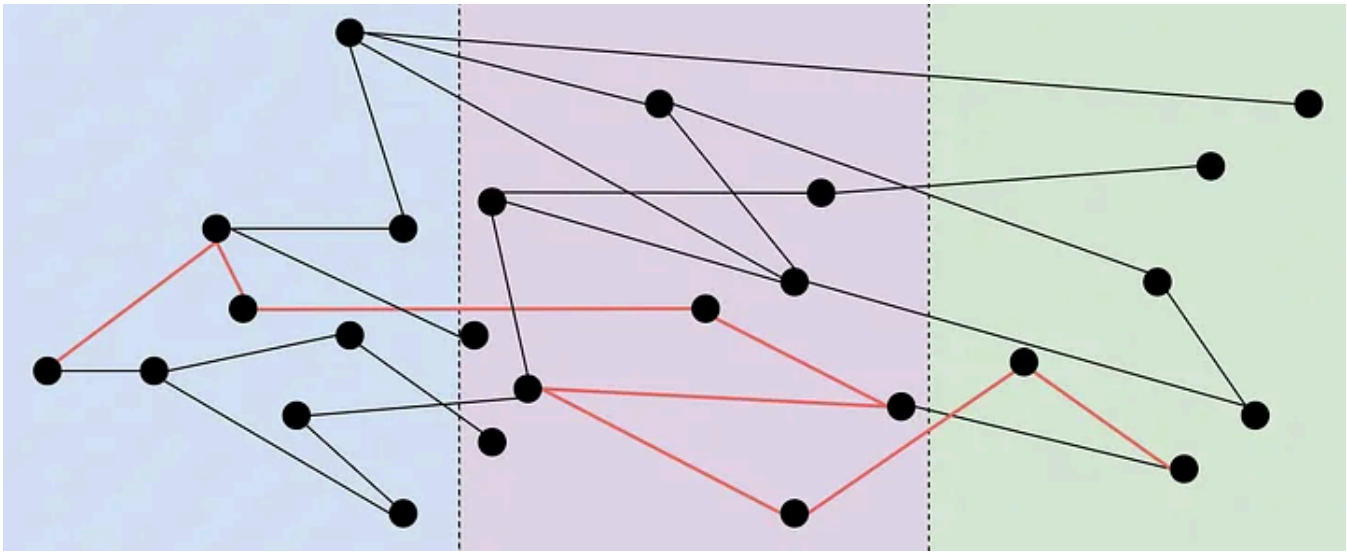

Fig 4.0: Partitioning the graph

Database

According to our understanding and requirements (high availability, high scalability, and fault tolerance), we can use **Cassandra** to store the **driver's last known location and the trip information after the trip has been completed**, and there will be **no updates** to it. We use Cassandra because the data we store is enormous and it increases continuously.

We can use a **MySQL** database to store trip information **while it's in progress**. We use MySQL for **in-progress trips for frequent updates** since the trip information is

relational, and it needs to be consistent across tables.

Schema

Riders	
Rider_ID:	INT
Name	VARCHAR
Email:	VARCHAR
Photo:	VARCHAR
Phone:	INT

Drivers	
Driver_ID:	INT
Name:	VARCHAR
Email:	VARCHAR
Photo:	VARCHAR
Phone:	INT
Vehicle_type:	VARCHAR
Vehicle_name:	VARCHAR
Vehicle_number:	INT

Driver_location	
Driver_ID:	INT
Old latitude:	DECIMAL
Old longitude:	DECIMAL
New latitude:	DECIMAL
New longitude:	DECIMAL

Trips	
Trip_ID:	INT
Rider_ID:	INT
Driver_ID:	INT
Location:	DECIMAL
ETA:	INT
status:	VARCHAR

Fig 5.0: The storage schema

Evaluation

Availability

Our system is highly available. We used WebSocket servers. If a user gets disconnected, the session is recreated via a load balancer with a different server. We've used multiple replicas of our databases with a primary-secondary replication model. We have the Cassandra database, which provides highly available services and no single point of failure. We used a CDN, cache, and load balancers, which increase the availability of our system.

Scalability

Our system is highly scalable. We used many independent services so that we can scale these services horizontally, independent of each other as per our needs. We used quadtrees for searching by dividing the map into smaller segments, which shortens our search space. We used a CDN, which increases the capacity to handle more users. We also used a NoSQL database, Cassandra, which is horizontally scalable. Additionally, we used load balancers, which improve speed by distributing read workload among different servers.

Reliability

Our system is highly reliable. The trip can continue even if the rider's or driver's connection is broken. This is achieved by using their phones as local storage. The use of multiple WebSocket servers ensures smooth, nearly real-time operations. If any of the servers fail, the user is able to reconnect with another server. We also used redundant copies of the servers and databases to ensure that there's no single point of failure. Our services are decoupled and isolated, which eventually increases the reliability. Load balancers help move the requests away from any failed servers to healthy ones.

Consistency

We used storage like MySQL to keep our data consistent globally. Moreover, our system does synchronous replication to achieve strong consistency. Because of a limited number of data writers and viewers for a trip (rider, driver, some internal services), the usage of traditional databases doesn't become a bottleneck. Also, data sharding is easier in this scenario.

Uber System Design | Software Architecture | Software Architect | System Design Interview | Distributed Systems

Edit profile

Written by Suresh Podeti

1.2K Followers

Aspiring Software Architect | SDE III (Golang) @JungleeGames| Ex-Byju's | Co-founder @Rupant Tech. | Ex-Synopsys | I.I.T Kharagpur | 7+ Years of Experience

Recommended from Medium